JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 25, 1617-1639 (2009)

A Semantic-Based Protocol for Concurrency Control in DOM Database Systems

KUEN-FANG JEA[1], TSUI-PING CHANG[1,2] AND SHIH-YING CHEN[3]
[1]*Department of Computer Science and Engineering*
National Chung Hsing University
Taichung, 402 Taiwan
[2]*Department of Business Administration*
Ling Tung University
Taichung, 408 Taiwan
[3]*Department of Computer Science and Information Engineering*
National Taichung Institute of Technology
Taichung, 404 Taiwan

Providing efficient access to XML documents is crucial, as XML has become the most important technique to exchange data in WWW. DOM is a popular object-oriented user interface to manipulate XML documents. Several concurrency control protocols have been proposed for DOM by analyzing the read/write behaviors of DOM operations. However, none of them exploit the semantics of DOM operations for enhancing concurrency. Semantics were introduced in object databases to develop concurrency control protocols. And this research is motivated by the success of this approach on object databases. In this paper, we analyze the commutativity relationship between DOM operations and propose a new semantic-based protocol for DOM, namely the SCD protocol. SCD not only allows non-serializable schedules to be executed, but also preserves the correctness of the resulting schedules. Our simulation results show that SCD outperforms other DOM-based protocols in its higher throughput and shorter response time. There are two major contributions in this paper. First, the semantics of DOM operations are analyzed formally. Second, based on the semantic analysis, a new way to design DOM-based concurrency control protocol is presented.

Keywords: DOM, XML, semantics, commutativity, concurrency control, database systems

1. INTRODUCTION

XML has enjoyed its wide popularity as a standard data exchange format for Internet applications, and databases that store XML documents have thus become increasingly important. Many areas of applications such as Finance, Medicine, and Science represent their document data in XML and exchange them via the Internet. Since documents in these areas are usually large, XML database management systems (XDBMSs) [21, 22] are helpful to support efficient access to these documents.

Concurrency control [4, 12, 14-16, 20] is one of the most important techniques providing efficient access in database systems. It improves system performance by concurrently executing multiple transactions. To this end, lock-based, timestamp-based, validation-based, and semantic-based concurrency control protocols have been proposed. Lock-

Received October 17, 2007; revised June 10, 2008; accepted August 26, 2008.
Communicated by Jorng-Tzong Horng.

based protocols [4, 12, 14, 16, 20] require that transactions request locks on data items before accessing them. Under timestamp-based protocols [4, 20], conflicting read and write operations are executed in timestamp order. Validation-based protocols [4, 15, 20] assume most operations in DBMSs are read operations. Under this type of protocols, DBMSs definitely finish transactions in the read phase and possibly rollback some of them in the validation phase if they do not pass conflict validations. In contrast, semantic-based protocols [3, 7, 13, 17, 18] have been proposed for object-oriented database systems (OODBMSs). They utilize the semantics of operations for allowing more operations to be executed concurrently.

Traditional concurrency control protocols [2, 3, 5, 7, 13] are not optimized for two features of XML documents. First, XML documents contain both structure and data, which must be maintained consistent during the execution of transactions. However, traditional concurrency control protocols do not consider document structures. Therefore, database operations on document structures are not optimized for high concurrency. Second, new operations (other than the traditional read and write operations) are introduced into XML, especially the DOM Application Program Interface (API). DOM [10] defines logical structures of XML documents and provides operations in its API for manipulating XML documents. However, traditional concurrency control protocols based on read/write conflicts are not tailored to these DOM operations. Therefore, new concurrency control protocols are required for enhancing XDBMS's performance.

To deal with DOM operations, several concurrency control protocols [8, 9] have been proposed. [8, 9] analyzed the behavior of read and write operations in DOM and proposed new lock models together with their compatibility matrix. These protocols are based on the serializability concept to determine conflicts between operations. Although they may successfully increase concurrency, there remains room of concurrency enhancement from the perspective of semantic-based approach. In fact, there are cases that, even if two DOM operations (such as the *NodeValue* and *ReplaceChild* operations on the same XML element) conflict under the protocols [8, 9], swapping the execution order of the two operations do not result in an inconsistent database state (*i.e.*, the values of manipulated XML elements are identical even if the two operations are swapped).

Semantic-based concurrency control protocols [3, 7, 13, 17, 18] have been proposed for OODBMSs. These protocols can provide more concurrency by determining whether two operations *commute* at the semantic level. Two operations commute if their execution orders do not produce different results, such as their returned object structures or values. By using the *commutativity* concept from operations' semantics, operations can run concurrently even though they are conflicting under those non-semantic-based protocols (*e.g.*, the two-phase locking protocol [20]). For example, on a queue data structure, operation *enqueue* adds an item to the back of the queue, while *dequeue* removes an item from the front of the queue. The two operations conflict under the two-phase locking protocol since they both modify the items in the queue. However, by analyzing the semantics of both operations, we may find that they commute and can be executed concurrently if the queue contains more than one item. The returned items in the queue are identical despite the execution orders of *enqueue* and *dequeue*.

Likewise, in XDBMSs the semantics of DOM operations can be explored to increase concurrency by determining which pairs of operations commute. To the best of our knowledge, however, no research at present addresses the commutativity issue of

DOM operations for concurrency enhancement. Accordingly, in this paper, by analyzing DOM operations' semantics and defining the commutativity relationship between DOM operations, we develop an efficient semantic-based concurrency control protocol, namely *SCD*, for DOM database systems. We also prove the correctness of SCD in this paper.

Two major contributions differentiate this research from others. First, we have analyzed DOM operations from the semantic perspective. Since none of current research exploits DOM operations in this way, the analysis result is useful for researchers to design new concurrency control techniques for XDBMSs. Second, we have developed a semantic-based concurrency-control protocol SCD. It shows that more DOM operations can be executed concurrently and higher concurrency is thus achieved.

This paper is organized as follows. Section 2 explains the DOM model, and section 3 analyzes the commutativity relationship of DOM operations. Section 4 proposes the new SCD protocol and proves the correctness of SCD schedules. Section 5 shows our simulation results, while section 6 reviews the related work. Finally, section 7 concludes this study.

2. PRELIMINARIES

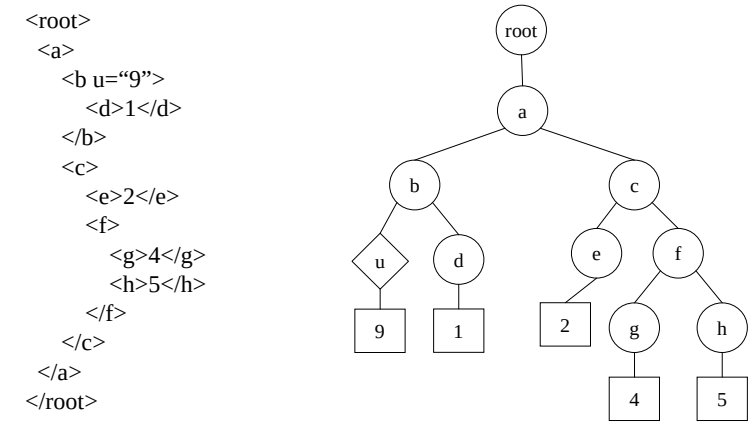
In this section, we describe the DOM model and classify DOM operations. Symbols in this research are defined as well.

2.1 Document Object Model (DOM)

The DOM model is suggested by the W3C [10]. It defines a document model to represent XML documents together with DOM operations to manipulate them. The DOM document model abstracts a document as a tree, namely XML tree. The root of an XML document is named *document element* in the XML tree, while elements, attributes, and texts in an XML document are mapped to nodes in the XML tree. Fig. 1 shows an example of XML document (a) and its corresponding XML tree (b). In Fig. 1 (b), the element nodes, attribute nodes, and text nodes are denoted by circles, rhombuses, and rectangles, respectively.

DOM defines several interfaces such as the element, characterdata, and node interfaces, to access elements, attributes, and texts in XML documents [10]. The element interface manipulates the attributes of elements in XML documents, while the characterdata interface updates the texts of elements or attributes. In addition, the node interface treats the elements, texts, and attributes in XML documents as nodes and defines operations to access them. The operations in the node interface read/write/insert/delete nodes in XML trees, while those in the element and characterdata interfaces read/write/insert/delete attributes and texts, respectively, in XML documents. The corresponding operations in the three interfaces have identical access behaviors, except that the objects (*i.e.*, nodes, texts, and attributes) operated by them are different. Therefore, only the operations in the node interface are analyzed in this paper. Nevertheless, the results of this semantic analysis can be applied equally well to the element and characterdata interfaces.

The operations in the node interface navigate and manipulate (*i.e.*, insert, update or delete) nodes in XML trees. Programmers can use the read operations (*e.g.*, *FirstChild*,



(a) An XML document. (b) The corresponding XML tree.

Fig. 1. An example of XML document.

NextSibling, *LastChild*, and *PreviousSibling*) to navigate nodes and use the write operations (*e.g.*, *InsertBefore*, *AppendChild*, *RemoveChild*, and *ReplaceChild*) to manipulate nodes. To navigate nodes, a reference node x with a read operation is indicated. For example, the expression "*x.FirstChild*" navigates the first child node of x. To modify nodes, a reference node x, an x's child node y and a new node n_1, with a write operation (*i.e.*, *InsertBefore*, *AppendChild*, and *ReplaceChild*), are indicated. The operation "*x.InsertBefore*(y, n_1)" inserts a new node n_1 in front of y, the operation "*x.AppendChild*(n_1)" appends a new last child node n_1 of x, while the operation "*x.ReplaceChild*(y, n_1)" removes all descendant nodes of y and replaces node y with a new node n_1. Finally, to delete nodes, a reference node x and a subtree rooted by x's child node y are indicated with a delete operation. For example, the operation "*x.RemoveChild*(y)" deletes node y and all of its descendant nodes.

2.2 Classification of DOM Operations

DOM operations (in the node interface) are classified into *observers* and *modifiers* for our semantic analysis purpose. An observer is a shared operation that reads nodes' values or navigates nodes in an XML tree, while a modifier is an exclusive operation that modifies nodes' values or document structures. As listed in Table 1, observers include *FirstChild* (denoted by **FC**), *NextSibling* (**NS**), *LastChild* (**LC**), *PreviousSibling* (**PS**),

Table 1. Classification of DOM operations.

Observers and denotations	Modifiers and denotations
FirstChild (**FC**)	*InsertBefore* (**IB**)
NextSibling (**NS**)	*RemoveChild* (**MC**)
LastChild (**LC**)	*AppendChild* (**AC**)
PreviousSibling (**PS**)	*ReplaceChild* (**RC**)
NodeName (**NN**)	
NodeValue (**NV**)	

NodeName (**NN**), and *NodeValue* (**NV**). Modifiers include *InsertBefore* (**IB**), *RemoveChild* (**MC**), *AppendChild* (**AC**), and *ReplaceChild* (**RC**). Among these operations, observers **NN** and **NV** read a node's name and value, respectively, while **FC**, **LC**, **NS**, and **PS** navigate a node's first child, last child, next sibling, and previous sibling, respectively. Modifiers **IB** and **AC** insert new nodes into an XML tree, and **MC** and **RC** respectively delete and replace nodes.

2.3 Observation

We have made the following observation on the semantics of DOM operations regarding the concurrency enhancement of DOM transactions. Two DOM operations may be executed concurrently in the following cases. Two observers can be executed concurrently, since they do not modify documents. On the other hand, in case at least one of the two operations is a modifier, the operations' semantics are helpful to decide if they can be executed concurrently. For example, in Fig. 1 modifier **AC** appends a new last child node of node *e*, while **IB** inserts a new node before an existing *e*'s child node. These two modifiers can be executed concurrently on the same node *e*, since their semantics indicate that the modified XML tree's structures are identical despite their execution orders. Therefore, exploiting the semantics of DOM operations is useful in increasing the concurrency of DOM transactions. We shall analyze DOM operations' semantics and develop an efficient protocol for XDBMSs based on the analysis.

2.4 Symbols

Symbols used in this paper are defined as follows. A DOM transaction T_i consists of a sequence of operations which are classified into observers and modifiers. We use notation $C_{i,m}$ to indicate the node operated by an operation $O_{i,m}$, which is the mth DOM operation in T_i. An observer in T_i is represented by $C_{i,m}.O_{i,m}$, while a modifier is represented by $C_{i,m}.O_{i,m}(N_{i,m}, R_{i,m})$, $C_{i,m}.O_{i,m}(N_{i,m})$, or $C_{i,m}.O_{i,m}(R_{i,m})$. $R_{i,m}$ denotes an existing child node of $C_{i,m}$, and $N_{i,m}$ denotes the new child node of $C_{i,m}$ to be inserted by $O_{i,m}$. For any observer $O_{i,m}$ (*i.e.*, **FC**, **LC**, **NN**, **NV**, **PS**, or **NS**), $C_{i,m}.O_{i,m}$ denotes that $C_{i,m}$ is read by $O_{i,m}$. $C_{i,m}.\mathbf{IB}(N_{i,m}, R_{i,m})$ indicates that modifier **IB** inserts $N_{i,m}$ in front of $R_{i,m}$, and $C_{i,m}.\mathbf{RC}(N_{i,m}, R_{i,m})$ indicates that modifier **RC** replaces $N_{i,m}$ with $R_{i,m}$, respectively. $C_{i,m}.\mathbf{AC}(N_{i,m})$ denotes that **AC** appends $N_{i,m}$ to $C_{i,m}$'s last child node, and $C_{i,m}.\mathbf{MC}(R_{i,m})$ denotes that **MC** deletes $R_{i,m}$ and all of $R_{i,m}$'s descendant nodes, respectively. Table 2 illustrates these symbols of DOM operations.

For example, in Fig. 1 the observer "*a.LastChild*" and modifier "*c.InsertBefore*(n_1, *f*)" in transaction T_1 can be represented by "$C_{1,1}.O_{1,1}$" and "$C_{1,2}.O_{1,2}(N_{1,2}, R_{1,2})$", respectively, where $C_{1,1}$ is node *a*, $C_{1,2}$ is node *c*, $O_{1,1}$ is observer **LC**, $O_{1,2}$ is modifier **IB**, $N_{1,2}$ is the new node n_1, and $R_{1,2}$ is node *f*. As a result, $T_1 = <a.\mathbf{LC}, c.\mathbf{IB}(n_1, f)>$.

3. SEMANTIC ANALYSIS OF DOM OPERATIONS

This paper considers the semantics of DOM operations (as listed in Table 1) on nodes and subtrees in XML trees. Each operation is classified into either a *node* or a *subtree operation* according to whether it operates on a node or a subtree. Node operations

Table 2. Symbols of DOM operations.

Symbols	Descriptions
$C_{i,m}$	the node operated by operation $O_{i,m}$
$R_{i,m}$	an existing child node of $C_{i,m}$
$N_{i,m}$	a new child node of $C_{i,m}$ to be inserted by $O_{i,m}$
$C_{i,m}.O_{i,m}$	the node $C_{i,m}$ is read by $O_{i,m}$
$C_{i,m}.O_{i,m}(N_{i,m}, R_{i,m})$	the new node $N_{i,m}$ is inserted before node $R_{i,m}$ by $O_{i,m}$, and the node $R_{i,m}$ is replaced with the new node $N_{i,m}$ by $O_{i,m}$
$C_{i,m}.O_{i,m}(N_{i,m})$	the node $N_{i,m}$ is appended to node $C_{i,m}$'s last child node
$C_{i,m}.O_{i,m}(R_{i,m})$	node $R_{i,m}$ and all its descendant nodes are deleted by $O_{i,m}$

include the **FC**, **LC**, **PS**, **NS**, **NN**, **NV, IB**, and **AC** operations, while subtree operations include the **MC** and **RC** operations. If operation $O_{i,m}$ is a node operation, $C_{i,m}$ is the node operated by $O_{i,m}$; otherwise, $C_{i,m}$ is the root node of a subtree operated by $O_{i,m}$. Operations' semantics consider the state changes of the operated nodes, including nodes' element value, element order, and their subtree structure. Node operations change nodes' element values and orders, while subtree operations change nodes' subtree structures. By considering the state changes of the operated nodes involved in two operations, the two operations may be executed concurrently without having one of them wait for the other (if switching their execution order always results in the same state).

The following terms S_x, V_x, E_x, and N_x are used to define the state of node x, and function $t(O_{i,m}, S_x)$ represents the state changes of node x operated by operation $O_{i,m}$ in transaction T_i. (Note that for simplicity, here we use x and z to represent nodes $C_{i,m}$ and $R_{i,m}$ respectively as defined in section 2.4, and we also use the $\mathbf{IB}_z$, $\mathbf{RC}_z$, and $\mathbf{MC}_z$ operations to represent the reference node z operated by the **IB**, **RC**, and **MC** operations, respectively.) State $S_x = (V_x, E_x, N_x)$ denotes the state of node x:

- V_x represents a tag name of an element node x or a text value of a text node x. It is a string enclosed by a pair of double quotes.
- E_x represents the sequence of x's child nodes from its first child node x_1 to its last child node x_n, and it is an ordered set denoted by $<x_1, x_2, \ldots, x_n>$.
- N_x represents the breadth-first search sequence of the nodes in x's subtree except node x itself, and it is an ordered set denoted by $<x_1, x_2, \ldots, x_n, x_{n+1}, \ldots>$.
- * is the symbol attached to a node x to indicate that the x's value or its child node has been read by an observer.
- $S_x' = t(O_{i,m}, S_x)$ where t is the state transformation function denoting that the new state S_x' is transformed by operation $O_{i,m}$ from the original state S_x.

For example, in Fig. 1 the c.**FC** operation reads node c's first child node. Before the **FC** operation is executed, the node c's state S_c = ("c", <e, f>, <e, f, 2, g, h, 4, 5>). After the execution of the **FC** operation, t(**FC**, S_c) = ("c", <e*, f>, <e*, f, 2, g, h, 4, 5>). Hereafter, we use the notation "$O_{i,m} < O_{j,n}$" to denote the execution order that operation $O_{i,m}$ precedes operation $O_{j,n}$ for simplicity.

Definition 1 Let S_x and S_y be the states of nodes x and y respectively before operations

$O_{i,m}$ and $O_{j,n}$ in transactions T_i and T_j. $O_{i,m}$ and $O_{j,n}$ commute if $t(O_{i,m}, t(O_{j,n}, S_x)) = t(O_{j,n}, t(O_{i,m}, S_x))$ and $t(O_{i,m}, t(O_{j,n}, S_y)) = t(O_{j,n}, t(O_{i,m}, S_y))$.

Definition 1 defines the commutativity relationship of two DOM operations $O_{i,m}$ and $O_{j,n}$ based on the state change of two XML nodes being operated. In Definition 1, the functions $t(O_{j,n}, t(O_{i,m}, S_x))$ and $t(O_{i,m}, t(O_{j,n}, S_x))$ respectively return the resulting states of node x after the two execution orders "$O_{i,m} < O_{j,n}$" and "$O_{j,n} < O_{i,m}$". The equality $t(O_{i,m}, t(O_{j,n}, S_x)) = t(O_{j,n}, t(O_{i,m}, S_x))$ means the identical resulting states of node x after both execution orders. Similarly, the equality $t(O_{i,m}, t(O_{j,n}, S_y)) = t(O_{j,n}, t(O_{i,m}, S_y))$ requires the identical states of node y after both execution orders. In general, the more the DOM operations commute, the higher concurrency the XDBMS achieves.

According to Definition 1, the commutativity of any two operations in Table 1 is analyzed. Lemmas 1 to 5 show the commutativity of observers and modifiers on nodes. On the other hand, Lemma 6 shows the commutativity of two subtree operations, and Lemmas 7 and 8 present the commutativity of a subtree operation and a node operation. These lemmas cover all of the possible commuting operations in DOM's node interface.

Lemma 1 For any two observers $O_{i,m}$, $O_{j,n} \in \{$**FC**, **NS**, **LC**, **PS**, **NN**, **NV**$\}$ in transactions T_i and T_j respectively, if they operate on the same node, then $O_{i,m}$ and $O_{j,n}$ commute.

Proof: We prove the lemma by showing that the states of the node being operated are identical after both execution orders of "$O_{i,m} < O_{j,n}$" and "$O_{j,n} < O_{i,m}$". Let $O_{i,m}$ and $O_{j,n}$ operate on node k and k's initial state be $S_k = (V_k, E_k, N_k)$. After the execution order "$O_{i,m} < O_{j,n}$", k's state becomes $t(O_{j,n}, t(O_{i,m}, S_k))$. Since $O_{i,m}$ is an observer on node k, it reads k's value or child nodes; that is, $t(O_{i,m}, S_k) = S_k' = (V_k^*, E_k^*, N_k)$. Thus, $t(O_{j,n}, t(O_{i,m}, S_k)) = t(O_{j,n}, S_k')$. Since $O_{j,n}$ is also an observer reading k's value or child nodes and V_k^*, E_k^* in S_k' already have the label "*" (marked as being read), V_k^* and E_k^* in S_k' need not be labeled by $O_{j,n}$ again and thus $t(O_{j,n}, S_k') = S_k'$. As a result, k's final state is $t(O_{j,n}, t(O_{i,m}, S_k)) = S_k'$.

Similarly, after the execution order "$O_{j,n} < O_{i,m}$", k's final state is $t(O_{i,m}, t(O_{j,n}, S_k)) = t(O_{i,m}, S_k') = S_k'$. Since both execution orders result in the same state for k, according to Definition 1, operations $O_{i,m}$ and $O_{j,n}$ commute. ❑

Lemma 2 For an observer $O_{i,m} \in \{$**NS**, **PS**, **NN**, **NV**$\}$ and a modifier $O_{j,n} \in \{$**IB**$_z$, **AC**$\}$ in transactions T_i and T_j respectively, if they operate on the same node, $O_{i,m}$ and $O_{j,n}$ commute.

Proof: Let $O_{i,m}$ and $O_{j,n}$ operate on node k and a k's child node z. k's initial state S_k = ("k", <k_1, k_2, …, z, …, k_n>, <k_1, k_2, …, z, …, k_n, k_{n+1}, …>). There are two cases, depending on whether $O_{j,n}$ is **IB**$_z$ or **AC**, that need to be analyzed.

Case 1: Suppose $O_{j,n}$ is **IB**$_z$. After "$O_{i,m} < O_{j,n}$", k's state becomes $t(O_{j,n}, t(O_{i,m}, S_k))$. Since $O_{i,m}$ is an observer reading k's value, the value "k" is labeled with the symbol "*" and $t(O_{i,m}, S_k) = S_k'$ = ("k"*, <k_1, k_2, …, z, …, k_n>, <k_1, k_2, …, z, …, k_n, k_{n+1}, …>). Thus, $t(O_{j,n}, t(O_{i,m}, S_k)) = t(O_{j,n}, S_k')$. Since $O_{j,n}$ inserts a new child node, say f, before node z, S_k' is changed into S_k'' = ("k"*, <k_1, k_2, …, f, z, …, k_n>, <k_1, k_2, …, f, z, …, k_n, k_{n+1}, …>), and

k's final state becomes $t(O_{j,n}, t(O_{i,m}, S_k)) = t(O_{j,n}, S_k') = S_k''$.

Alternatively, after "$O_{j,n} < O_{i,m}$", k's final state is $t(O_{i,m}, t(O_{j,n}, S_k))$. Similarly, since $O_{j,n}$ inserts the new node f before node z, S_k is changed into $t(O_{j,n}, S_k) = S_k' = (\text{“}k\text{”}, <k_1, k_2, \ldots, f, z, \ldots, k_n>, <k_1, k_2, \ldots, f, z, \ldots, k_n, k_{n+1}, \ldots>)$. Thus, $t(O_{i,m}, t(O_{j,n}, S_k)) = t(O_{i,m}, S_k')$. Further, since $O_{i,m}$ reads k's value, it changes S_k' into $t(O_{i,m}, S_k') = S_k'' = (\text{“}k\text{”}*, <k_1, k_2, \ldots, f, z, \ldots, k_n>, <k_1, k_2, \ldots, f, z, \ldots, k_n, k_{n+1}, \ldots>)$. Therefore $t(O_{i,m}, t(O_{j,n}, S_k)) = t(O_{i,m}, S_k') = S_k''$. In conclusion, since k's final states after both the execution orders "$O_{i,m} < O_{j,n}$" and "$O_{j,n} < O_{i,m}$" are identical, according to Definition 1, $O_{i,m}$ and $O_{j,n}$ commute.

Case 2: Suppose $O_{j,n}$ is **AC**. After "$O_{i,m} < O_{j,n}$", k's state becomes $t(O_{j,n}, t(O_{i,m}, S_k))$. Since $O_{i,m}$ reads k's value, S_k is changed into $t(O_{i,m}, S_k) = S_k' = (\text{“}k\text{”}*, <k_1, k_2, \ldots, z, \ldots, k_n>, <k_1, k_2, \ldots, z, \ldots, k_n, k_{n+1}, \ldots>)$. Thus, $t(O_{j,n}, t(O_{i,m}, S_k)) = t(O_{j,n}, S_k')$. Since $O_{j,n}$ inserts a new last child node, say f, into node k, S_k' is changed into $S_k'' = (\text{“}k\text{”}*, <k_1, k_2, \ldots, z, \ldots, k_n, f>, <k_1, k_2, \ldots, z, \ldots, k_n, f, k_{n+1}, \ldots>)$, and k's final state becomes $t(O_{j,n}, t(O_{i,m}, S_k)) = t(O_{j,n}, S_k') = S_k''$.

Alternatively, after "$O_{j,n} < O_{i,m}$", k's final state is $t(O_{i,m}, t(O_{j,n}, S_k))$. Similarly, since $O_{j,n}$ inserts the new last child node f, S_k is changed into $S_k' = (\text{“}k\text{”}, <k_1, k_2, \ldots, z, \ldots, k_n, f>, <k_1, k_2, \ldots, z, \ldots, k_n, f, k_{n+1}, \ldots>)$. Further, since $O_{i,m}$ reads k's value, it changes S_k' into $t(O_{i,m}, S_k') = S_k'' = (\text{“}k\text{”}*, <k_1, k_2, \ldots, z, \ldots, k_n, f>, <k_1, k_2, \ldots, z, \ldots, k_n, f, k_{n+1}, \ldots>)$. Therefore $t(O_{i,m}, t(O_{j,n}, S_k)) = t(O_{i,m}, S_k') = S_k''$. In conclusion, since k's final states after both "$O_{i,m} < O_{j,n}$" and "$O_{j,n} < O_{i,m}$" are identical, according to Definition 1, $O_{i,m}$ and $O_{j,n}$ commute.

Based on Cases 1 and 2, we thus prove this lemma. ❑

Lemma 3 An observer **FC** and a modifier **AC** commute if they operate on the same node k and k has at least one child node.

Proof: Let k's initial state $S_k = (\text{“}k\text{”}, <k_1, k_2, \ldots, k_n>, <k_1, k_2, \ldots, k_n, k_{n+1}, \ldots>)$. After "**FC** < **AC**", k's state is $t(\mathbf{AC}, t(\mathbf{FC}, S_k))$. Since **FC** reads the first child node of k, $t(\mathbf{FC}, S_k) = S_k' = (\text{“}k\text{”}, <k_1*, k_2, \ldots, k_n>, <k_1*, k_2, \ldots, k_n, k_{n+1}, \ldots>)$. Assume modifier **AC** inserts a new last child node, say f, into node k. S_k' is changed into $S_k'' = (\text{“}k\text{”}, <k_1*, k_2, \ldots, k_n, f>, <k_1*, k_2, \ldots, k_n, f, k_{n+1}, \ldots>)$, and k's final state $t(\mathbf{AC}, t(\mathbf{FC}, S_k)) = t(\mathbf{AC}, S_k') = S_k''$.

Alternatively, after "**AC** < **FC**", k's final state is $t(\mathbf{FC}, t(\mathbf{AC}, S_k))$. Similarly, since **AC** inserts the new last child node f into node k, S_k is changed into $S_k' = (\text{“}k\text{”}, <k_1, k_2, \ldots, k_n, f>, <k_1, k_2, \ldots, k_n, f, k_{n+1}, \ldots>)$. Further, since **FC** reads the first child node of k, $t(\mathbf{FC}, S_k') = S_k'' = (\text{“}k\text{”}, <k_1*, k_2, \ldots, k_n, f >, <k_1*, k_2, \ldots, k_n, f, k_{n+1}, \ldots)$. Therefore, $t(\mathbf{FC}, t(\mathbf{AC}, S_k)) = t(\mathbf{FC}, S_k') = S_k''$.

Since k's states after both orders "**FC** < **AC**" and "**AC** < **FC**" are identical, according to Definition 1, **FC** and **AC** commute. ❑

Lemma 4 An observer **LC** and a modifier $\mathbf{IB}_z$ commute if they operate on the same node k and k has at least one child node.

Proof: Let k's initial state $S_k = (\text{“}k\text{”}, <k_1, k_2, \ldots, z, \ldots, k_n>, <k_1, k_2, \ldots, z, \ldots, k_n, k_{n+1}, \ldots>)$. After "$\mathbf{LC} < \mathbf{IB}_z$", k's state is $t(\mathbf{IB}_z, t(\mathbf{LC}, S_k))$. Since **LC** is an observer reading the last child node of k, $t(\mathbf{LC}, S_k) = S_k' = (\text{“}k\text{”}, <k_1, k_2, \ldots, z, \ldots, k_n*>, <k_1, k_2, \ldots, z, \ldots, k_n*,$

k_{n+1}, …>). And since $\mathbf{IB}_z$ inserts a new child node, say f, before node z, S_k' is changed into $S_k'' = (\text{“}k\text{”}, <k_1, k_2, \ldots, f, z, \ldots, k_n^*>, <k_1, k_2, \ldots, f, z, \ldots, k_n^*, k_{n+1}, \ldots>)$, and k's final state $t(\mathbf{IB}_z, t(\mathbf{LC}, S_k)) = t(\mathbf{IB}_z, S_k') = S_k''$.

Alternatively, after "$\mathbf{IB}_z < \mathbf{LC}$", k's final state is $t(\mathbf{LC}, t(\mathbf{IB}_z, S_k))$. Similarly, since $\mathbf{IB}_z$ inserts the new node f before node z, S_k is changed into $S_k' = (\text{“}k\text{”}, <k_1, k_2, \ldots, f, z, \ldots, k_n>, <k_1, k_2, \ldots, f, z, \ldots, k_n, k_{n+1}, \ldots>)$. Further, since **LC** reads the last child node of k, $t(\mathbf{LC}, t(\mathbf{IB}_z, S_k)) = t(\mathbf{LC}, S_k') = S_k'' = (\text{“}k\text{”}, <k_1, k_2, \ldots, f, z, \ldots, k_n^*>, <k_1, k_2, \ldots, f, z, \ldots, k_n^*, k_{n+1}, \ldots>)$.

Since k's final states after both "$\mathbf{LC} < \mathbf{IB}_z$" and "$\mathbf{IB}_z < \mathbf{LC}$" are identical, according to Definition 1, **LC** and $\mathbf{IB}_z$ commute. ❑

Lemma 5 Two modifiers **AC** and $\mathbf{IB}_z$ commute if they operate on the same node k and k has at least one child node.

Proof: Let k's initial state $S_k = (\text{“}k\text{”}, <k_1, k_2, \ldots, z, \ldots, k_n>, <k_1, k_2, \ldots, z, \ldots, k_n, k_{n+1}, \ldots>)$. After "$\mathbf{AC} < \mathbf{IB}_z$", k's state is $t(\mathbf{IB}_z, t(\mathbf{AC}, S_k))$. Assume **AC** appends a new last child node, say f, into node k. S_k is changed into $S_k' = (\text{“}k\text{”}, <k_1, k_2, \ldots, z, \ldots, k_n, f>, <k_1, k_2, \ldots, z, \ldots, k_n, f, k_{n+1}, \ldots>)$, and $t(\mathbf{IB}_z, t(\mathbf{AC}, S_k)) = t(\mathbf{IB}_z, S_k')$. Further, assume $\mathbf{IB}_z$ inserts a new child node, say g, before node z. S_k' is changed into $S_k'' = (\text{“}k\text{”}, <k_1, k_2, \ldots, g, z, \ldots, k_n, f>, <k_1, k_2, \ldots, g, z, \ldots, k_n, f, k_{n+1}, \ldots>)$, and k's final state $t(\mathbf{IB}_z, t(\mathbf{AC}, S_k)) = t(\mathbf{IB}_z, S_k') = S_k''$.

Alternatively, after "$\mathbf{IB}_z < \mathbf{AC}$", k's final state is $t(\mathbf{AC}, t(\mathbf{IB}_z, S_k))$. Similarly, since $\mathbf{IB}_z$ inserts node g before node z, S_k is changed into state $S_k' = (\text{“}k\text{”}, <k_1, k_2, \ldots, g, z, \ldots, k_n>, <k_1, k_2, \ldots, g, z, \ldots, k_n, k_{n+1}, \ldots>)$. Further, since **AC** appends node f to k's children, S_k' is changed into $S_k'' = (\text{“}k\text{”}, <k_1, k_2, \ldots, g, z, \ldots, k_n, f>, <k_1, k_2, \ldots, g, z, \ldots, k_n, f, k_{n+1}, \ldots>)$. Therefore, $t(\mathbf{AC}, t(\mathbf{IB}_z, S_k)) = t(\mathbf{AC}, S_k') = S_k''$.

Since k's states after both "$\mathbf{AC} < \mathbf{IB}_z$" and "$\mathbf{IB}_z < \mathbf{AC}$" are identical, according to Definition 1, **AC** and $\mathbf{IB}_z$ commute. ❑

Lemma 6 For a modifier $O_{i,m} \in \{\mathbf{MC}_z, \mathbf{RC}_z\}$ on node g and another modifier $O_{j,n} \in \{\mathbf{MC}_u, \mathbf{RC}_u\}$ on node h, if there does not exist an ancestor-descendant relationship between nodes g and h, $O_{i,m}$ and $O_{j,n}$ commute.

Proof: Let the initial states of nodes g and h be $S_g = (\text{“}g\text{”}, <g_1, g_2, \ldots, z, \ldots, g_n>, <g_1, g_2, \ldots, z, \ldots, g_n, g_{n+1}, z_1, \ldots, z_n, \ldots>)$ and $S_h = (\text{“}h\text{”}, <h_1, h_2, \ldots, u, \ldots, h_n>, <h_1, h_2, \ldots, u, \ldots, h_n, h_{n+1}, u_1, \ldots, u_n, \ldots>)$ respectively, where $z_1, \ldots, z_n$ are all of z's descendant nodes and $u_1, \ldots, u_n$ are all of u's descendant nodes. Since the ancestor-descendant relationship does not exist between g and h, the operation $O_{i,m} \in \{\mathbf{MC}_z, \mathbf{RC}_z\}$ on g neither deletes h nor affects h's child nodes. Similarly, the operation $O_{j,n} \in \{\mathbf{MC}_u, \mathbf{RC}_u\}$ on h neither deletes g nor affects g's child nodes. There are four cases in total, depending on $O_{i,m}$ is $\mathbf{MC}_z$ or $\mathbf{RC}_z$, and $O_{j,n}$ is $\mathbf{MC}_u$ or $\mathbf{RC}_u$. Case 1: $O_{i,m} = \mathbf{MC}_z$, and $O_{j,n} = \mathbf{MC}_u$; Case 2: $O_{i,m} = \mathbf{MC}_z$, and $O_{j,n} = \mathbf{RC}_u$; Case 3: $O_{i,m} = \mathbf{RC}_z$, and $O_{j,n} = \mathbf{RC}_u$; and Case 4: $O_{i,m} = \mathbf{RC}_z$, and $O_{j,n} = \mathbf{MC}_u$. However, Case 4 is similar to Case 2 and can be analyzed likewise. Thus, only Cases 1, 2, and 3 are proven here.

Case 1: $O_{i,m} = \mathbf{MC}_z$, and $O_{j,n} = \mathbf{MC}_u$. After "$\mathbf{MC}_z < \mathbf{MC}_u$", the states of g and h become $t(\mathbf{MC}_z, S_g)$ and $t(\mathbf{MC}_u, S_h)$, respectively. Since $\mathbf{MC}_z$ deletes node z and all of z's descen-

dant nodes, S_g is changed into S_g' = ("g", <g_1, g_2, …, g_n>, <g_1, g_2, …, g_n, g_{n+1}, …>) by deleting nodes z, z_1, …, z_n. Similarly, since $\mathbf{MC}_u$ deletes node u and all of u's descendant nodes, S_h is changed into S_h' = ("h", <h_1, h_2, …, h_n>, <h_1, h_2, …, h_n, h_{n+1}, …>) by deleting nodes u, u_1, …, u_n. As a result, g's state $t(\mathbf{MC}_z, S_g) = S_g'$, and h's state $t(\mathbf{MC}_u, S_h) = S_h'$.

Alternatively, after "$\mathbf{MC}_u$<$\mathbf{MC}_z$", since $\mathbf{MC}_z$ and $\mathbf{MC}_u$ delete nodes z and u and their descendant nodes, S_g is changed into S_g' = (V_g, <g_1, g_2, …, g_n>, <g_1, g_2, …, g_n, g_{n+1}, …>) and S_h is changed into S_h' = ("h", <h_1, h_2, …, h_n>, <h_1, h_2, …, h_n, h_{n+1}, …>). As a result, g's state $t(\mathbf{MC}_z, S_g) = S_g'$, and h's state $t(\mathbf{MC}_u, S_h) = S_h'$. In conclusion, since g and h have the same final states after both execution orders of "$\mathbf{MC}_z < \mathbf{MC}_u$" and "$\mathbf{MC}_u < \mathbf{MC}_z$", by Definition 1, $\mathbf{MC}_z$ and $\mathbf{MC}_u$ commute.

Case 2: $O_{i,m} = \mathbf{MC}_z$, and $O_{j,n} = \mathbf{RC}_u$. After "$\mathbf{MC}_z < \mathbf{RC}_u$", the states of g and h become $t(\mathbf{MC}_z, S_g)$ and $t(\mathbf{RC}_u, S_h)$, respectively. Since $\mathbf{MC}_z$ deletes node z and all of z's descendant nodes, S_g is changed into S_g' = ("g", <g_1, g_2, …, g_n>, <g_1, g_2, …, g_n, g_{n+1}, …>) by deleting nodes z, z_1, …, z_n. Similarly, since $\mathbf{RC}_u$ replaces node u with a new node, say a, S_h is changed into S_h' = ("h", <h_1, h_2, …, a, …, h_n>, <h_1, h_2, …, a, …, h_n, h_{n+1}, …>) by deleting nodes u_1, …, u_n and replacing u with a. As a result, g's state $t(\mathbf{MC}_z, S_g) = S_g'$, and h's state $t(\mathbf{RC}_u, S_h) = S_h'$.

Alternatively, after "$\mathbf{RC}_u < \mathbf{MC}_z$", since $\mathbf{MC}_z$ deletes node z and all of z's descendant nodes, and $\mathbf{RC}_u$ deletes all of u's descendant nodes and replaces u with a, S_g is changed into S_g' = ("g", <g_1, g_2, …, g_n>, <g_1, g_2, …, g_n, g_{n+1}, …>) and S_h is changed into S_h' = ("h", <h_1, h_2, …, a, …, h_n>, <h_1, h_2, …, a, …, h_n, h_{n+1}, …>). As a result, g's state $t(\mathbf{MC}_z, S_g) = S_g'$, and h's state $t(\mathbf{RC}_u, S_h) = S_h'$. In conclusion, since g and h respectively have the same final states after both execution orders of "$\mathbf{MC}_z < \mathbf{RC}_u$" and "$\mathbf{RC}_u < \mathbf{MC}_z$", by Definition 1, $\mathbf{MC}_z$ and $\mathbf{RC}_u$ commute.

Case 3: $O_{i,m} = \mathbf{RC}_z$, and $O_{j,n} = \mathbf{RC}_u$. After "$\mathbf{RC}_z < \mathbf{RC}_u$", the states of g and h become $t(\mathbf{RC}_z, S_g)$ and $t(\mathbf{RC}_u, S_h)$, respectively. Since $\mathbf{RC_z}$ replaces node z with node, say a, S_g is changed into S_g' = ("g", <g_1, g_2, …, a, …, g_n, g_{n+1}, …>, <g_1, g_2, …, a, …, g_n, g_{n+1}, …>) by deleting nodes z_1, …, z_n and replacing z with a. Similarly, since $\mathbf{RC}_u$ replaces node u with a node, say b, S_h is changed into S_h' = ("h", h_1, h_2, …, b, …, h_n>, <h_1, h_2, …, b, …, h_n, h_{n+1}, …>) by deleting nodes u_1, …, u_n and replacing u with b. As a result, g's state $t(\mathbf{RC}_z, S_g) = S_g'$ and h's state $t(\mathbf{RC}_u, S_h) = S_h'$.

Alternatively, after "$\mathbf{RC}_u < \mathbf{RC}_z$", since $\mathbf{RC}_z$ and $\mathbf{RC}_u$ replace nodes z and u with nodes a and b respectively, S_g is changed into S_g' = ("g", <g_1, g_2, …, a, …, g_n>, <g_1, g_2, …, a, …, g_n, g_{n+1}, …>) and S_h is changed into S_h' = ("h", "h", h_1, h_2, …, b, …, h_n>, <h_1, h_2, …, b, …, h_n, h_{n+1}, …>). As a result, g's state $t(\mathbf{RC}_z, S_g) = S_g'$ and h's state $t(\mathbf{RC}_u, S_h) = S_h'$. In conclusion, since g and h respectively have the same final states after both "$\mathbf{RC}_z < \mathbf{RC}_u$" and "$\mathbf{RC}_u < \mathbf{RC}_z$", by Definition 1, $\mathbf{RC}_z$ and $\mathbf{RC}_u$ commute.

Based on Cases 1, 2, and 3, we thus prove this lemma. ❑

Lemma 7 For an observer $O_{i,m} \in \{\mathbf{NS}, \mathbf{PS}, \mathbf{NN}, \mathbf{NV}, \mathbf{FC}, \mathbf{LC}\}$ on node g and a modifier $O_{j,n} \in \{\mathbf{RC_z}, \mathbf{MC_z}\}$ on node h, if there does not exist an ancestor-descendant relationship between nodes g and h, $O_{i,m}$ and $O_{j,n}$ commute.

Proof: Let the initial states of g and h be $S_g = (V_g, E_g, N_g)$ and $S_h = (\text{“}h\text{”}, <h_1, h_2, \ldots, z, \ldots, h_n>, <h_1, h_2, \ldots, z, \ldots, h_n, h_{n+1}, z_1, \ldots, z_n, \ldots>)$ respectively, where nodes $z_1, \ldots, z_n$ are all of z's descendant nodes. Since the ancestor-descendant relationship does not exist between g and h, the operation $O_{j,n}$ on h does not affect g's state. And since the operation $O_{i,m}$ on node g, it does not affect h's state. There are two cases, depending on whether $O_{j,n}$ is $\mathbf{RC}_z$ or $\mathbf{MC}_z$, that need to be analyzed.

Case 1: $O_{j,n} = \mathbf{RC}_z$. After "$O_{i,m} < O_{j,n}$", the states of g and h become $t(O_{i,m}, S_g)$ and $t(O_{j,n}, S_h)$, respectively. Since $O_{i,m}$ is an observer reading g's value or child nodes, it changes S_g into $t(O_{i,m}, S_g) = S_g' = (V_g^*, E_g^*, N_g)$. And since $O_{j,n}$ replaces node z with a node, say a, S_h is changed into $S_h' = (\text{“}h\text{”}, <h_1, h_2, \ldots, a, \ldots, h_n>, <h_1, h_2, \ldots, a, \ldots, h_n, h_{n+1}, \ldots>)$ by deleting nodes $z_1, \ldots, z_n$ and replacing z with a. Thus, $t(O_{i,m}, S_g) = S_g'$ and $t(O_{j,n}, S_h) = S_h'$.

Alternatively, after "$O_{j,n} < O_{i,m}$", the states of g and h are $t(O_{i,m}, S_g)$ and $t(O_{j,n}, S_h)$ respectively. Similarly, since $O_{i,m}$ is an observer on g, $t(O_{i,m}, S_g) = S_g' = (V_g^*, E_g^*, N_g)$. Further, since $O_{j,n}$ replaces node z with a new node a, S_h is changed into $S_h' = (\text{“}h\text{”}, <h_1, h_2, \ldots, a, \ldots, h_n>, <h_1, h_2, \ldots, a, \ldots, h_n, h_{n+1}, \ldots>)$. Thus, $t(O_{i,m}, S_g) = S_g'$ and $t(O_{j,n}, S_h) = S_h'$. In conclusion, since g and h respectively have the same final states after "$O_{i,m} < O_{j,n}$" and "$O_{j,n} < O_{i,m}$", by Definition 1, $O_{i,m}$ and $O_{j,n}$ commute.

Case 2: $O_{j,n} = \mathbf{MC}_z$. After "$O_{i,m} < O_{j,n}$", the states of g and h become $t(O_{i,m}, S_g)$ and $t(O_{j,n}, S_h)$, respectively. Since $O_{i,m}$ is an observer on g, it changes $t(O_{i,m}, S_g) = S_g' = (V_g^*, E_g^*, N_g)$. And since $O_{j,n}$ deletes node z and all of z's descendant nodes, S_h is changed into state $S_h' = (\text{“}h\text{”}, <h_1, h_2, \ldots, h_n>, <h_1, h_2, \ldots, h_n, h_{n+1}, \ldots>)$ by deleting nodes $z, z_1, \ldots, z_n$. Thus, $t(O_{i,m}, S_g) = S_g'$ and $t(\mathbf{MC}_z, S_h) = S_h'$.

Alternatively, after "$O_{j,n} < O_{i,m}$", the states of g and h are $t(O_{i,m}, S_g)$ and $t(O_{j,n}, S_h)$ respectively. Similarly, since $O_{i,m}$ is an observer on g, $t(O_{i,m}, S_g) = S_g' = (V_g^*, E_g^*, N_g)$. Further, since $O_{j,n}$ deletes node z, S_h is changed into $S_h' = (\text{“}h\text{”}, <h_1, h_2, \ldots, h_n>, <h_1, h_2, \ldots, h_n, h_{n+1}, \ldots>)$. Thus, $t(O_{i,m}, S_g) = S_g'$ and $t(O_{j,n}, S_h) = S_h'$. In conclusion, since g and h respectively have the same final states after "$O_{i,m} < O_{j,n}$" and "$O_{j,n} < O_{i,m}$", by Definition 1, $O_{i,m}$ and $O_{j,n}$ commute.

Based on Cases 1 and 2, we thus prove this lemma. ❑

Lemma 8 For a modifier $O_{i,m} \in \{\mathbf{IB}_z, \mathbf{AC}\}$ on node g and another modifier $O_{j,n} \in \{\mathbf{RC}_u, \mathbf{MC}_u\}$ on node h, if there does not exist an ancestor-descendant relationship between nodes g and h, $O_{i,m}$ and $O_{j,n}$ commute.

Proof: Let the initial states of nodes g and h be $S_g = (\text{“}g\text{”}, <g_1, g_2, \ldots, z, \ldots, g_n>, <g_1, g_2, \ldots, z, \ldots, g_n, g_{n+1}, \ldots>)$ and $S_h = (\text{“}h\text{”}, <h_1, h_2, \ldots, u, \ldots, h_n>, <h_1, h_2, \ldots, u, \ldots, h_n, h_{n+1}, u_1, \ldots, u_n, \ldots>)$, respectively, where $u_1, \ldots, u_n$ are all of u's descendant nodes. Since the ancestor-descendant relationship does not exist between g and h, the operation $O_{j,n}$ on h does not affect g's state. And since the operation $O_{i,m}$ operates on node g, it does not affect h's state. There are four cases that need to be analyzed, depending on whether $O_{i,m}$ is $\mathbf{IB}_z$ or $\mathbf{AC}$, and $O_{j,n}$ is $\mathbf{MC}_u$ or $\mathbf{RC}_u$.

Case 1: $O_{i,m} = \mathbf{IB}_z$ and $O_{j,n} = \mathbf{MC}_u$. After "$\mathbf{IB}_z < \mathbf{MC}_u$", the states of g and h become

$t(\mathbf{IB}_z, S_g)$ and $t(\mathbf{MC}_u, S_h)$, respectively. Since $\mathbf{IB}_z$ inserts a new node, say f, before node z, S_g is changed into S_g' = ("g", <g_1, g_2, …, f, z, …, g_n>, <g_1, g_2, …, f, z, …, g_n, g_{n+1}, …>). Similarly, since $\mathbf{MC}_u$ deletes a child node u of node h, S_h is changed into S_h' = ("h", <h_1, h_2, …, h_n>, <h_1, h_2, …, h_n, h_{n+1}, …>) by deleting nodes u, u_1, …, u_n. As a result, g's state $t(\mathbf{IB}_z, S_g) = S_g'$ and h's state $t(\mathbf{MC}_u, S_h) = S_h'$.

Alternatively, after "$\mathbf{MC}_u < \mathbf{IB}_z$", since $\mathbf{IB}_z$ inserts a new node f, before z and $\mathbf{MC}_u$ deletes node u, S_g is changed into S_g' = ("g", <g_1, g_2, …, f, z, …, g_n>, <g_1, g_2, …, f, z, …, g_n, g_{n+1}, …>) and S_h is changed into S_h' = ("h", <h_1, h_2, …, h_n>, <h_1, h_2, …, h_n, h_{n+1}, …>) by deleting nodes u, u_1, …, u_n. As a result, g's state $t(\mathbf{IB}_z, S_g) = S_g'$ and h's state $t(\mathbf{MC}_u, S_h) = S_h'$. In conclusion, since g and h respectively have the same final states after both execution orders of "$\mathbf{IB}_z < \mathbf{MC}_u$" and "$\mathbf{MC}_u < \mathbf{IB}_z$", by Definition 1, $\mathbf{IB}_z$ and $\mathbf{MC}_u$ commute.

Case 2: $O_{i,m} = \mathbf{IB}_z$ and $O_{j,n} = \mathbf{RC}_u$. After "$\mathbf{IB}_z < \mathbf{RC}_u$", the states of g and h become $t(\mathbf{IB}_z, S_g)$ and $t(\mathbf{RC}_u, S_h)$, respectively. Since $\mathbf{IB}_z$ inserts a new node, say f, before node z, S_g is changed into S_g' = ("g", <g_1, g_2, …, f, z, …, g_n>, <g_1, g_2, …, f, z, …, g_n, g_{n+1}, …>). Similarly, since $\mathbf{RC}_u$ replaces node u with a node, say a, S_h is changed into S_h' = ("h", <h_1, h_2, …, a, …, h_n>, <h_1, h_2, …, a, …, h_n, h_{n+1}, …>) by deleting nodes u_1, …, u_n and replacing u with a. As a result, g's state $t(\mathbf{IB}_z, S_g) = S_g'$ and h's state $t(\mathbf{RC}_u, S_h) = S_h'$.

Alternatively, after "$\mathbf{RC}_u < \mathbf{IB}_z$", since $\mathbf{IB}_z$ inserts a new node f before z and $\mathbf{RC}_u$ replaces node u with node a, S_g is changed into S_g' = ("g", <g_1, g_2, …, f, z, …, g_n>, <g_1, g_2, …, f, z, …, g_n, g_{n+1}, …>) and S_h is changed into S_h' = ("h", <h_1, h_2, …, a, …, h_n>, <h_1, h_2, …, a, …, h_n, h_{n+1}, …>). As a result, g's state $t(\mathbf{IB}_z, S_g) = S_g'$ and h's state $t(\mathbf{RC}_u, S_h) = S_h'$. In conclusion, since g and h respectively have the same final states after both execution orders of "$\mathbf{IB}_z < \mathbf{RC}_u$" and "$\mathbf{RC}_u < \mathbf{IB}_z$", by Definition 1, $\mathbf{IB}_z$ and $\mathbf{RC}_u$ commute.

Case 3: $O_{i,m} = \mathbf{AC}$ and $O_{j,n} = \mathbf{MC}_u$. After "$\mathbf{AC} < \mathbf{MC}_u$", the states of g and h become $t(\mathbf{AC}, S_g)$ and $t(\mathbf{MC}_u, S_h)$, respectively. Since $\mathbf{AC}$ inserts a new last child node, say f, into node g, S_g is changed into S_g' = ("g", <g_1, g_2, …, z, …, g_n, f>, <g_1, g_2, …, f, z, …, g_n, f, g_{n+1}, …>). Similarly, since $\mathbf{MC}_u$ deletes nodes u, u_1, …, u_n, S_h is changed into S_h' = ("h", <h_1, h_2, …, h_n>, <h_1, h_2, …, h_n, h_{n+1}, …>). As a result, g's state $t(\mathbf{AC}, S_g) = S_g'$ and h's state $t(\mathbf{MC}_u, S_h) = S_h'$.

Alternatively, after "$\mathbf{MC}_u < \mathbf{AC}$", since $\mathbf{AC}$ inserts a new last child node f and $\mathbf{MC}_u$ deletes node u, S_g is changed into S_g' = ("g", <g_1, g_2, …, z, …, g_n, f>, <g_1, g_2, …, f, z, …, g_n, f, g_{n+1}, …>) and S_h is changed into S_h' = ("h", <h_1, h_2, …, h_n>, <h_1, h_2, …, h_n, h_{n+1}, …>). As a result, g's state $t(\mathbf{AC}, S_g) = S_g'$ and h's state $t(\mathbf{MC}_u, S_h) = S_h'$. In conclusion, since g and h respectively have the same final states after both execution orders of "$\mathbf{AC} < \mathbf{MC}_u$" and "$\mathbf{MC}_u < \mathbf{AC}$", by Definition 1, $\mathbf{AC}$ and $\mathbf{MC}_u$ commute.

Case 4: $O_{i,m} = \mathbf{AC}$ and $O_{j,n} = \mathbf{RC}_u$. After "$\mathbf{AC} < \mathbf{RC}_u$", the states of g and h become $t(\mathbf{AC}, S_g)$ and $t(\mathbf{RC}_u, S_h)$, respectively. Since $\mathbf{AC}$ inserts a new last child node, say f, into node g, S_g is changed into S_g' = ("g", <g_1, g_2, …, z, …, g_n, f>, <g_1, g_2, …, f, z, …, g_n, f, g_{n+1}, …>). Similarly, since $\mathbf{RC}_u$ replaces node u with a node, say a, S_h is changed into S_h' = ("h", <h_1, h_2, …, a, …, h_n>, <h_1, h_2, …, a, …, h_n, h_{n+1}, …>). As a result, g's state $t(\mathbf{AC}, S_g) = S_g'$ and h's state $t(\mathbf{RC}_u, S_h) = S_h'$.

Alternatively, after "$\mathbf{RC}_u < \mathbf{AC}$", since $\mathbf{AC}$ inserts a new last child node f and $\mathbf{RC}_u$

replaces node u with node a, S_g is changed into $S_g' = (\text{“}g\text{”}, <g_1, g_2, \ldots, z, \ldots, g_n, f>, <g_1, g_2, \ldots, f, z, \ldots, g_n, f, g_{n+1}, \ldots>)$ and S_h is changed into $S_h' = (\text{“}h\text{”}, <h_1, h_2, \ldots, a, \ldots, h_n>, <h_1, h_2, \ldots, a, \ldots, h_n, h_{n+1}, \ldots>)$. As a result, g's state $t(\mathbf{AC}, S_g) = S_g'$ and h's state $t(\mathbf{RC}_u, S_h) = S_h'$. In conclusion, since g and h respectively have the same final states after both execution orders of "$\mathbf{AC} < \mathbf{RC}_u$" and "$\mathbf{RC}_u < \mathbf{AC}$", by Definition 1, **AC** and $\mathbf{RC}_u$ commute.

Based on Cases 1, 2, 3, and 4, we thus prove this lemma. ❑

Table 3 illustrates the commutativity between any two operations, and it indeed covers all of the possible commuting operations in Table 1. The number n in each entry in Table 3 indicates that the commutativity of the two corresponding operations is governed by the lemma n. Note that the symbol "X" in some entries denotes no commutativity relationship between the two corresponding operations.

Table 3. Commutativity relationships of DOM operations.

	FC	NS	LC	PS	NN	NV	IB	AC	MC	RC
FC	1	1	1	1	1	1	X	3	7	7
NS	1	1	1	1	1	1	2	2	7	7
LC	1	1	1	1	1	1	4	X	7	7
PS	1	1	1	1	1	1	2	2	7	7
NN	1	1	1	1	1	1	2	2	7	7
NV	1	1	1	1	1	1	2	2	7	7
IB	X	2	4	2	2	2	X	5	8	8
AC	3	2	X	2	2	2	5	X	8	8
MC	7	7	7	7	7	7	8	8	6	6
RC	7	7	7	7	7	7	8	8	6	6

4. SCD PROTOCOL

In this section, we propose a semantic-based protocol SCD for DOM databases. The correctness of SCD schedules is proved, and its implementation issues are also analyzed.

4.1 SCD Protocol Rules

SCD ensures concurrent execution of operations based on their commutativity. For checking commutativity, we associate a stamp for each operation $O_{i,m}$ on node k in transaction T_i. The stamp, denoted by $(T_i, O_{i,m}[k])$, includes the transaction T_i's identity, the operation type of $O_{i,m}$ and the node k operated by $O_{i,m}$. If the execution of operation $O_{i,m}$ is allowed according to the SCD protocol, the stamp $(T_i, O_{i,m}[k])$ is recorded in the system.

Suppose that $O_{i,m}$ and $O_{j,n}$ are two operations in transactions T_i and T_j, respectively, and both operate on the same node k. If operation $O_{i,m}$ has been executed and the stamp $(T_i, O_{i,m}[k])$ is recorded in the system, then $O_{j,n}$ must observe the following rules before being executed.

Rule 1: The commutativity between $O_{j,n}$ and $O_{i,m}$ is checked according to Lemmas 1 to 8.
(1) If the operation type of $O_{j,n}$ and that of $O_{i,m}$ recorded in the stamp (T_i, $O_{i,m}[k]$) commute, $O_{j,n}$ can be executed and the stamp (T_j, $O_{j,n}[k]$) is recorded in the system before the execution of $O_{j,n}$.
(2) Otherwise, $O_{j,n}$ can not be executed until the stamp (T_i, $O_{i,m}[k]$) is deleted.
Rule 2: Before recording, if the stamp (T_j, $O_{j,n}[k]$) has already been recorded in the system, then the stamp (T_j, $O_{j,n}[k]$) is not recorded again.
Rule 3: The stamp (T_j, $O_{j,n}[k]$) is deleted from the system after transaction T_j finishes.
Rule 4: T_j is aborted if a deadlock is detected.

Table 4. A possible schedule of transactions T_1 and T_2 under SCD.

Steps	T_1	T_2
1	*a*.FC	
2	*c*.LC	
3		*a*.FC
4		*c*.LC
5	*c*.IB(n_1, *e*)	
6	T_1 finishes	
7		*c*.FC
8		T_2 finishes

SCD works as follows. Consider two transactions T_1 and T_2 executing on the XML document in Fig. 1, where T_1 comprises three DOM operations *a*.**FC**, *c*.**LC** and *c*.**IB**(n_1, *e*), and T_2 comprises three operations *a*.**FC**, *c*.**LC**, and *c*.**FC**. Table 4 illustrates a possible schedule of T_1 and T_2 under SCD. At first, in steps 1 and 2, T_1's operations *a*.**FC** and *c*.**LC** are executed sequentially, and the stamps (T_1, **FC**[*a*]) and (T_1, **LC**[*c*]) are recorded in the system. Next, T_2's operations *a*.**FC** and *c*.**LC** are also executed respectively in steps 3 and 4, since T_1's *a*.**FC** and T_2's *a*.**FC** commute and T_1's *c*.**LC** and T_2's *c*.**LC** commute. Also, the stamps (T_2, **FC**[*a*]) and (T_2, **LC**[*c*]) are recorded. Further, T_1's *c*.**IB**(n_1, *e*) is executed in step 5 and stamp (T_1, **IB**[*c*]) is recorded, since T_2's *c*.**LC** commutes with T_1's *c*.**IB**(n_1, *e*). After T_1 finishes in step 6, the stamps (T_1, **FC**[*a*]), (T_1, **LC**[*c*]), and (T_1, **IB**[*c*]) are deleted. Note that T_2's operation *c*.**FC** in step 7 must wait till T_1 finishes, since it does not commute with T_1's *c*.**IB**(n_1, *e*). Finally, the stamps (T_2, **FC**[*a*]), (T_2, **LC**[*c*]), and (T_2, **FC**[*c*]) are all deleted after T_2 finishes in step 8. (Note that the stamp is designed for detecting the first pair of non-commuting operations between transactions. Once a stamp corresponding to some operation is recorded in the system, it is unnecessary to record the stamp again, as specified in Rule 2, while executing the same operation again since other operations not commuting (*i.e.*, conflicting) with this operation can be detected by the stamp already recorded in the system.)

Derived from the SCD protocol rules, Corollaries 1 and 2 show the features of SCD schedules. Corollary 1 illustrates that any SCD schedule has no deadlock, while Corollary 2 shows the execution order of any two transactions in an SCD schedule.

Corollary 1 Any SCD schedule has no deadlock.

Proof: It is directly from the Rule 4 of SCD protocol. ❑

Corollary 2 For any two transactions T_i and T_j in an SCD schedule H, if $O_{i,m}$ of T_i and $O_{j,n}$ of T_j is the first pair of non-commuting operations on some node k and "$O_{i,m} < O_{j,n}$", the operations after $O_{j,n}$ in T_j must wait until T_i finishes.

Proof: According to the Rule 1 of SCD protocol, the two non-commuting operations $O_{i,m}$ of T_i and $O_{j,n}$ of T_j can not be executed concurrently. Since "$O_{i,m} < O_{j,n}$", the stamp (T_i, $O_{i,m}[k]$) is already recorded in the system when $O_{j,n}$ is going to be executed. Therefore, by the Rule 1 (b) of SCD protocol, $O_{j,n}$ can not be executed until the stamp (T_i, $O_{i,m}[k]$) is deleted. However, the stamp (T_i, $O_{i,m}[k]$) can only be deleted when T_i finishes by the Rule 3 of SCD protocol. Hence, the operations after $O_{j,n}$ in T_j must wait until T_i finishes. We thus prove this corollary. ❑

4.2 Correctness of SCD Schedules

In this subsection, we prove the correctness of SCD schedules based on their final states. According to [6, 15, 19], a schedule is said *correct* if its final state is the same as that of a serial schedule. We shall prove that any schedule under SCD results in a final state identical to that from some serial schedule.

The two terms F_u and $Z_u(T_i, T_j)$ are used in this paper: F_u denotes the final state of the nodes (*i.e.*, the set of the states of the affected nodes) operated by the operations in a schedule H_u, and $Z_u(T_i, T_j)$ denotes the sequence (ordered by the execution order) of the first pair of non-commuting operations from transactions T_i and T_j in H_u. For example, if H_u is the schedule shown in Table 4, F_u represents the final state of nodes a and c after executing all operations in H_u. Namely, $F_u = \{S_a, S_c\}$ = {("a", <b, c>, <b, c, u, d, n_1, e, f, 9, 1, 2, g, h, 4, 5>), ("c", <n_1, e, f>, <n_1, e, f, 2, g, h, 4, 5>)}, and $Z_u(T_1, T_2)$ = <c.**IB**(n_1, e) of T_1, c.**FC** of T_2>.

Definition 2 (Equivalent schedules) Let H_a and H_b be two schedules with the same set of transactions. H_a and H_b are **equivalent** if and only if states F_a and F_b are identical.

According to Definition 2, the correctness of SCD schedules is analyzed. Lemma 9 shows two equivalent schedules have the same execution orders of non-commuting operations from any two transactions, while Lemmas 10 and 11 demonstrate that any SCD schedule is equivalent to some serial schedule. Theorem 1 then concludes that any SCD schedule is correct.

Lemma 9 Two schedules H_a and H_b under SCD with the same set of transactions are equivalent if $Z_a(T_i, T_j) = Z_b(T_i, T_j)$, where $Z_a(T_i, T_j)$ and $Z_b(T_i, T_j)$ respectively are the sequences of the first pair of non-commuting operations from any two transactions T_i and T_j in H_a and H_b.

Proof: Consider any two transactions T_i and T_j in H_a and H_b. Let **Q** be the set of operations in H_a and H_b that come from only T_i and T_j. Suppose $Z_a(T_i, T_j) = Z_b(T_i, T_j) = <O_{i,m}, O_{j,n}>$. Let **S1** be the set of operations in T_i and those before $O_{j,n}$ in T_j. And let **S2** be the

set of operations in T_j after (and including) $O_{j,n}$. Thus, **Q** = **S1** ∪ **S2**. According to Rule 1 (a), the operations of T_i before (and including) $O_{i,m}$ and those of T_j in **S1** commute. Further, the operations of T_i after $O_{i,m}$ and those of T_j in **S1** must also commute; otherwise, a deadlock will occur and Corollary 1 is violated. Thus, T_i's and T_j's operations in **S1** commute, and by Definition 1 any execution order of them, including H_a and H_b, always results in the identical states of nodes. Let F_a' and F_b' respectively be the final states of the sub-schedules of H_a and H_b comprising only the operations in **S1**. Then, $F_a' = F_b'$. After that, concerning **Q**, H_a and H_b comprise only the operations in **S2**. Since these operations are all from T_j and can be executed in only one order, *i.e.*, the T_j's order, the sub-schedules of H_a and H_b in this part are identical. The final states F_a of H_a generated from F_a' and those F_b of H_b generated from F_b' are thus identical. According to Definition 2, H_a is equivalent to H_b. ❑

Lemma 10 Any SCD schedule comprising only two transactions is equivalent to a serial schedule comprising only the two transactions.

Proof: Let H_a and H_b be an SCD schedule and a serial schedule respectively both comprising the two transactions T_i and T_j. There are two cases, depending on whether operations in T_i and T_j commute or not.

Case 1: Suppose all operations in T_i and those in T_j commute. By Definition 1, any execution order of them including H_a and H_b always results in the identical final states of nodes. According to Definition 2, H_a is equivalent to H_b.

Case 2: Suppose that at least two operations from T_i and T_j respectively do not commute and $O_{i,m}$, $O_{j,n}$ are the first pair of such non-commuting operations. Assume that in H_a, $O_{i,m}$ appears before $O_{j,n}$; *i.e.*, $Z_a(T_i, T_j) = <O_{i,m}, O_{j,n}>$. Let H_b be the serial schedule of execution order "$T_i < T_j$". Since in H_b, all operations in T_i are executed before those in T_j, $O_{i,m}$ must appear before $O_{j,n}$ and $Z_b(T_i, T_j) = <O_{i,m}, O_{j,n}>$. Thus, $Z_a(T_i, T_j) = Z_b(T_i, T_j)$, and according to Lemma 9, H_a is equivalent to H_b. Similarly, if $O_{i,m}$ appears after $O_{j,n}$ in H_a, we let H_b be the serial schedule of execution order "$T_j < T_i$". For the same reason, $Z_a(T_i, T_j) = Z_b(T_i, T_j) = <O_{j,n}, O_{i,m}>$. Since a serial schedule can always be produced under the SCD protocol rules (*i.e.*, without violating the SCD protocol rules), H_a is equivalent to H_b according to Lemma 9.

Based on Cases 1 and 2, we thus prove this lemma. ❑

Corollary 3 For any SCD schedule H_a comprising only two transactions T_i and T_j, if $Z_a(T_i, T_j) = <O_{i,m}, O_{j,n}>$, it is equivalent to a serial schedule of execution order "$T_i < T_j$".

Proof: It is directly from the Case 2 proof of Lemma 10. ❑

Lemma 11 Any SCD schedule is equivalent to a serial schedule.

Proof: Let H_a be an SCD schedule comprising $(n + 1)$ transactions T_0, T_1, T_2, …, and T_n. ($n > 1$). According to Lemma 10 and Corollary 3, an SCD schedule H_d contains only two

transactions T_i and T_j is equivalent to the serial schedule with execution order decided by $Z_d(T_i, T_j)$, namely the sequence of the first pair of non-commuting operations in T_i and T_j. Thus, for every pair of transactions T_i and T_j in H_a, the serial execution orders of any two transactions among T_0, T_1, T_2, …, and T_n can be obtained. Since no deadlock is allowed in any SCD schedule according to Corollary 1, these serial execution orders of any two transactions can not result in a cyclic precedence of execution orders among T_0, T_1, T_2, …, and T_n. By a topological sorting of these serial execution orders, a schedule H_b with a linear execution order of transactions T_0, T_1, T_2, …, and T_n can be obtained. Since the execution order of every pair of transactions T_i and T_j in H_a is in accord with the linear execution order of T_i and T_j in H_b, $Z_a(T_i, T_j) = Z_b(T_i, T_j)$. According to Lemma 9, H_a and H_b are equivalent. We thus prove this lemma. ❑

Theorem 1 Any SCD schedule is correct.

Proof: Any SCD schedule is equivalent to some serial schedule by Lemma 11. By Definition 2, it has the same final state as that of the serial schedule. According to [6, 15, 19], the SCD schedule is correct. The theorem is thus proved. ❑

4.3 Implementation Issues

In this subsection, we discuss the implementation issues of SCD protocol, including the data structure for recording the stamps and the deadlock detection problem.

A data structure on each node is necessary to record the stamps on that node for efficiently checking commutativity between operations. For this purpose, we may associate with each node a linked list of stamps on that node and organize the heads of all linked lists (for all of the affected nodes) into a hash table. Suppose that an operation $O_{j,n}$ in transaction T_j is going to be executed on a node c. The linked list of stamps associated with c is located first via a hash function on c. $O_{j,n}$ and each operation $O_{i,m}$ of transaction T_i in c's stamp list are then checked to see if $O_{i,m}$ and $O_{j,n}$ commute. If they do, the identity of $O_{j,n}$ is inserted in c's stamp list. After T_i finishes, all the stamps regarding operations $O_{i,m}$ in T_i are deleted from c's list.

Another important issue is the deadlock problem. Deadlocks may occur in general since transactions may wait for each other if the operations in these transactions do not commute. If a deadlock happens in generating SCD schedules, the transaction leading to the deadlock must be aborted according to the Rule 4 of SCD protocol. To fulfill this requirement, traditional deadlock detection and recovery algorithms [1, 20] can be used. However, a costly deadlock-detection graph may be maintained. A more suitable method is to design a deadlock-free protocol without the need of deadlock detection. This is part of our future work.

5. EXPERIMENTAL RESULTS

This section illustrates our simulation study on the performance of SCD, OO2PL [9] and taDOM [8]. Our simulation experiments are conducted on the simulation platform in [11]. The simulation platform has the implemented classes to generate documents and

transactions for scheduling transactions under the SCD, OO2PL, and taDOM protocols. It can also store the evaluation results (*i.e.*, throughputs, response times, wait times, and number of aborted transactions) in its database system [11]. Table 5 lists the parameters and their settings in our two experiments on the simulation platform. The parameters *n*, *p*, *min*, and *max* describe the features of the generated documents, the *mpl* parameter denotes the maximum number of concurrent transactions in the system, and the parameters *t*, *w*, and *o* indicate the setting values of the generated transactions.

Table 5. Simulation parameters and settings.

Parameters	Descriptions	Settings
n	number of nodes in a document	425
p	document depth	5
min	minimal fan-out of node	4
max	maximal fan-out of node	6
t	number of transactions	300 ~ 1200
w	percentage of write transactions	10% ~ 50%
o	the number of operations in a transaction	3
mpl	maximal multi-programming level in the system	5

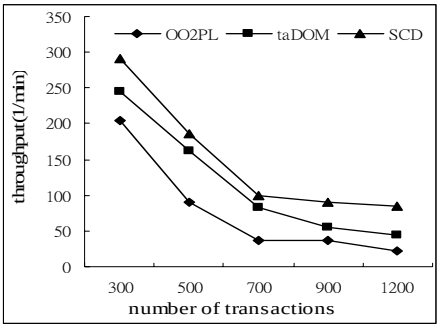


Fig. 2. Throughputs of SCD, taDOM and OO2PL.

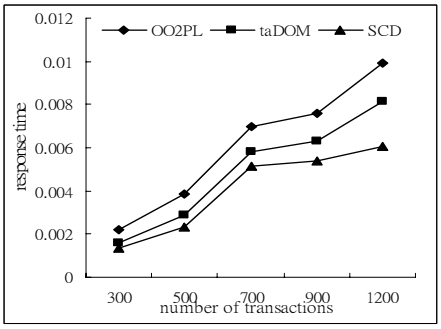


Fig. 3. Response times of SCD, taDOM, and OO2PL.

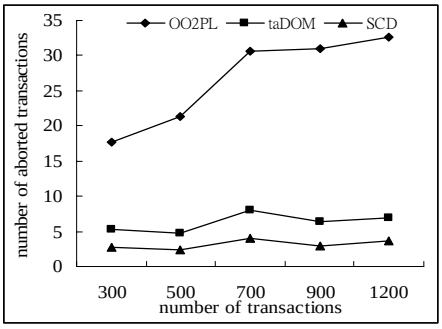


Fig. 4. Numbers of aborted transactions of SCD, taDOM and OO2PL.

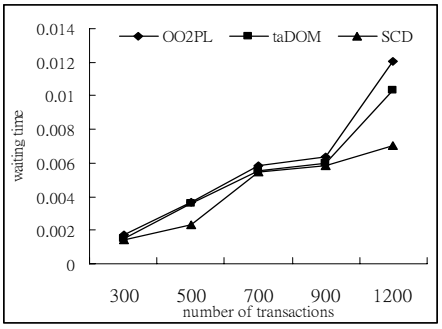


Fig. 5. Wait times of SCD, taDOM, and OO2PL.

The first experiment, with results shown in Figs. 2, 3, 4, and 5, observes the throughput, response time, number of aborted transactions, and wait time (Y-axis) of the three protocols under different number of transactions (X-axis). The parameter settings are: n = 425, p = 5, min = 4, max = 6, w = 30%, o = 3, and mpl = 5. In Fig. 2, SCD outperforms OO2PL and taDOM on the throughput. However, except t = 300 and t = 500, all the curves for OO2PL, taDOM, and SCD change slightly as the number of transactions increases. A possible reason is the control of the parameter mpl, which dominates the number of transactions being executed at the same time and thus dominates the throughputs. In Fig. 3, the response times of SCD, OO2PL and taDOM increase as the number of transactions increases and SCD also outperforms the other two. In Figs. 4 and 5, SCD has a lower aborted transaction rate and shorter wait time. The reason is that SCD considers operations' semantics and allows more operations to be executed concurrently. (We shall explain this point in section 6 below.)

The second experiment, with results shown in Figs. 6, 7, 8, and 9, measures throughput, response time, the number of aborted transactions, and wait time of the SCD, OO2PL and taDOM protocols under different percentages of exclusive operations (*i.e.*, modifiers **AC**, **RC**, **IB**, and **MC**). The parameter settings are: n = 425, p = 5, min = 4, max = 6, o = 3, t = 300, and mpl = 5. Figs. 6 and 7 show that SCD outperforms OO2PL and taDOM on the throughput and response time, while Figs. 8 and 9 show that SCD has less number

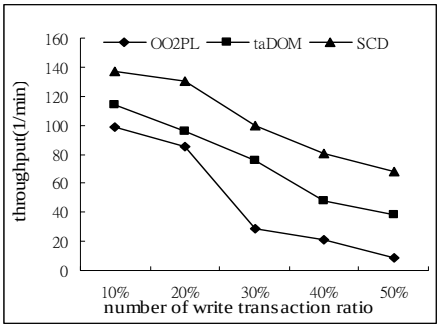


Fig. 6. Throughputs of SCD, taDOM and OO2PL.

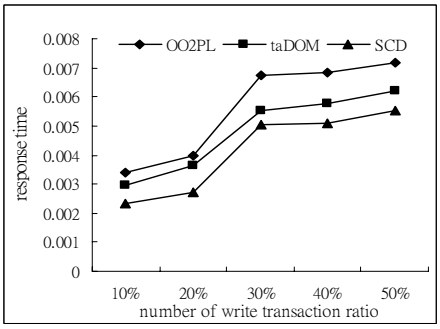


Fig. 7. Response times of SCD, taDOM and OO2PL.

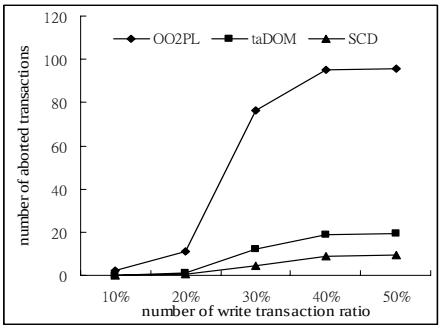


Fig. 8. Numbers of aborted transactions of SCD, taDOM and OO2PL.

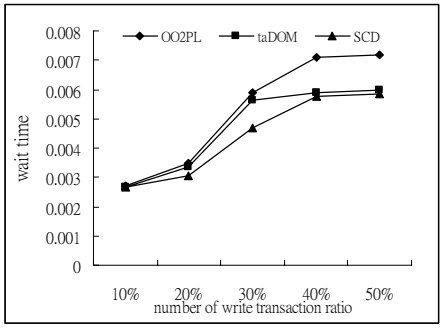


Fig. 9. Wait times of SCD, taDOM, and OO2PL.

of aborted transactions and shorter wait time than those in OO2PL and taDOM. The reason is that SCD allows some exclusive operations, such as **IB** and **AC**, to be executed concurrently, leading to a higher throughput, shorter response time/wait time, and lower aborted transaction rate.

In conclusion, the two experiments show that SCD has higher concurrency than OO2PL and taDOM. This is because by exploiting operations' semantics, some conflicts occurring in OO2PL and taDOM may be ignored in SCD. This point will be explained in section 6.

6. RELATED WORK

In this section, two types of DOM-based protocols [8, 9] are reviewed. The first type [9] includes the Doc2PL, NO2PL, Node2PL and OO2PL protocols. These protocols are different in lock granularity. Doc2PL locks the document root node and thus locks the entire XML tree. NO2PL locks the parent of the manipulated node to prevent the node from being visited by other transactions. Node2PL and OO2PL provide finer lock granularities which are the manipulated node itself and the pointers leading to the manipulated node respectively. In these protocols, [9] demonstrated that OO2PL outperforms others since it acquires only the locks on the necessary pointers of a node for operations.

The second type [8] includes a series of concurrency control protocols, namely taDOM, taDOM2, taDOM2+, taDOM3, and taDOM3+. These taDOM protocol family uses intention locks (*i.e.*, IR and IX locks) and navigation locks (*i.e.*, ER, EX, and EU locks) to lower down conflicts. Intention locks can increase concurrency since they do not lock nodes in the whole subtree, resulting in the concurrent execution of an update operation (*i.e.*, **IB** and **AC** operations) on a node with the other operation (*i.e.*, **MC** and **RC** operation) on nodes underneath that node in another transaction. On the other hand, navigation locks are used for navigation purpose (for example, by using **FC**, **LC**, **NS**, and **PS** operations).

One reason confirms that SCD may outperform OO2PL and taDOM. The concurrent execution of update and navigation operations is not allowed under both OO2PL and taDOM protocols, but it may be allowed under SCD. This is because the semantics of DOM operations are exploited in SCD. Through the semantics, SCD considers the correct state of XML nodes for allowing more operations to be executed simultaneously. For example, suppose that two transactions T_1 and T_2 manipulate the document in Fig. 1, where T_1 comprises three DOM operations *a*.**FC**, *c*.**FC,** and *c*.**IB**(n_1, *e*), and T_2 comprises two operations *a*.**FC** and *c*.**AC**(n_2). Also, suppose that the operations of T_1 have been executed and those of T_2 are going to be executed. Under both taDOM and OO2PL protocols, the operation *a*.**FC** in T_2 can be executed since node *a* can be read concurrently by operations *a*.**FC** in T_1 and *a*.**FC** in T_2. Alternatively, the *c*.**AC**(n_2) operation in T_2 can not be executed since it will modify node *c* which is already read by the *c*.**FC** operation in T_1. Therefore, the operation *c*.**AC**(n_2) of T_2 must wait until T_1 finishes. However, all operations in T_2 can be executed under SCD since the *a*.**FC** and *c*.**AC**(n_2) operations in T_2 commute with the *a*.**FC** and *c*.**FC** operations in T_1 respectively.

7. CONCLUSION

In this paper, we exploit the DOM operations' semantics to increase concurrency in the XDBMSs using the DOM API. By defining the commutativity of DOM operations, we propose a new semantic-based concurrency control protocol, namely SCD, to improve the performance of XDBMSs. We also prove that SCD is correct; that is, schedules under SCD always generate the same database states as some serial schedules do. Further, experimental results show that SCD outperforms both taDOM and OO2PL protocols on throughput and response time. Our future work includes designing a deadlock-free protocol without the need of deadlock detection, since deadlock may happen in the generation of SCD schedules.

REFERENCES

1. R. Agrawal, M. J. Carey, and D. J. DeWitt, "Deadlock detection is cheap," *SIGMOD Record*, Vol. 13, 1983, pp. 19-34.
2. B. Badrinath and K. Ramamrithan, "Semantics-based concurrency control: Beyond commutativity," *ACM Transactions on Database Systems*, Vol. 17, 1992, pp. 163-199.
3. C. Beeri, P. Bernstein, and N. Goodman, "A model for concurrency in nested transactions systems," *Journal of the ACM*, Vol. 1, 1989, pp. 240-269.
4. P. Bernstein, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, Boston, 1987.
5. A. A. Farrag and M. T. Özsu, "Using semantic knowledge of transactions to increase concurrency," *ACM Transactions on Database Systems*, Vol. 14, 1989, pp. 503-525.
6. M. J. Fischer, N. D. Griffeth, and N. A. Lynch, "Global states of a distributed system," *IEEE Transactions on Software and Engineering*, Vol. 8, 1982, pp. 198-202.
7. P. Goyal, T. S. Narayanan, and F. Sadri, "Concurrency control for object bases," *Journal of Information Systems*, Vol. 18, 1993, pp. 167-180.
8. M. P. Haustein and T. Härder, "Adjustable transaction isolation in XML database management systems," in *Proceedings of the 2nd International Conference on XSym*, 2004, pp. 173-188.
9. S. Helmer, C. Kanne, and G. Moerkotte, "Evaluating lock-based protocols for cooperation on XML documents," *SIGMOD Record*, Vol. 33, 2004, pp. 58-63.
10. A. L. Hors, P. L. Hegaret, and L. Wood, "Document object model (DOM) level 3 core specification version 1.0, World Wide Web consortium (W3C) recommendation," http://www.w3.org/TR/2004/REC-DOM-level-3-core-20040407/, 2004.
11. K. F. Jea, C. W. Cheng, and T. P. Chang, "An object-oriented simulation platform for XML-based concurrency control protocols," in *Proceedings of the 16th Workshop on Object-Oriented Technology and Application*, 2005, pp. 340-347.
12. K. F. Jea and S. Y. Chen, "A high concurrency XPath-based locking protocol for XML databases," *Information and Software Technology*, Vol. 48, 2006, pp. 708-716.
13. W. Jun, "Semantic-based locking technique in object-oriented databases," *Information and Software Technology*, Vol. 42, 2000, pp. 524-531.
14. H. F. Korth, "Locking primitives in a database system," *Journal of ACM*, Vol. 30,

1983, pp. 55-79.
15. H. T. Kung and C. H. Papadimitriou, "An optimality theory of concurrency control for databases," in *Proceedings of the ACM SIGMOD Conference on Management of Data*, 1979, pp. 116-126.
16. S. Y. Lee and R. L. Liou, "A multi-granularity locking model for concurrency control in object-oriented database systems," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 8, 1996, pp. 144-156.
17. P. Muth, T. C. Rakow, and G. Weikum, "Semantic concurrency control in object-oriented database systems," in *Proceedings of the IEEE International Conference on Data Engineering*, 1993, pp. 243-252.
18. R. F. Resende, D. Agrawal, and A. E. Abbadi, "Semantic locking in object-oriented database systems," *ACM Transactions on Databases Systems*, Vol. 29, 1994, pp. 388-402.
19. P. M. Schwarz and A. Z. Spector, "Synchronizing shared abstract types," *ACM Transactions on Computer Systems*, Vol. 2, 1984, pp. 223-250.
20. A. Silberschatz, H. Korth, and S. Sudarshan, *Database System Concepts*, 5th ed., McGraw-Hill, New York, 2006.
21. http://www.data-ex-machina.com/natix.html.
22. http://www.softwareag.com/corporate/products/tamino.

Kuen-Fang Jea (賈坤芳) received the Ph.D. degree in Computer Science and Engineering from the University of Michigan at Ann Arbor in 1989. He is an Associate Professor in the Department of Computer Science and Engineering, National Chung Hsing University (NCHU), Taiwan. Before joining NCHU, he was with Bellcore (Bell Communication Research). His major research interests include XML database system, database performance, data mining, data stream system and mobile computing. Dr. Jea is a member of Eta Kappa Nu, the ACM and the IEEE Computer Society.

Tsui-Ping Chang (張翠蘋) received the M.S. degree in Information Management from National Yunlin University of Science and Technology in 2000. Currently, she is a Ph.D. candidate in the department of Computer Science and Engineering, National Chung Hsing University. Since 2000, she was the faculty member of the Department of Business Administration, Kao Yuan University. In 2001, she joined the Department of Business Administration, Ling Tung University. Her research interests include XML database systems and object-oriented systems.

Shih-Ying Chen (陳世穎) received his Ph.D. degree in Computer Science from National Chung Hsing University in 2006. Currently, he is with both the faculty of the Computer Science and Information Engineering, and the Information Management departments of National Taichung Institute of Technology. His major research interests include XML database systems and mobile computing.